Robotics Assignment 2022 (COMS4045)

Nico Jared Govindsamy
Tristan Bookhan

Mapping

We used *gmapping* to map the environment. The image produced in *RViz* was then saved as *.png* file. We then applied several image processing techniques to transform the map image to a configuration space map that the path planner can use. The image processing techniques are outlined here:

1. Resize the map to an appropriate scale and size.
2. Convert the RGBA image to a grayscale image.
3. Perform binary thresholding on the image (Open spaces are True and obstacles are False).
4. Perform multiple iterates of morphological erosion on the image to make obstacles, like walls, thicker. This accounts for the thickness of the turtlebot and creates a configuration space kind of map where the robot uses safer paths that avoid obstacles.
5. Save the output to a text file that will be used by the path finding program.

These techniques are achieved in the *ProcessMap.py* script using the *skimage* library. (Cannot be run in ROS Singularity terminal!)

Path Planning

The method used for planning a path was Probabilistic Roadmap (PRM) with some minor additions. This worked effectively because there are large open spaces in part of the map which allows for points to be scattered in a lot of free spaces, therefore a route is likely to be found.

The main file, which gives instructions to the robot, calls the $FindPath()$ function. This function takes in the start and goal coordinates plus the radius and resolution properties for PRM. It then reads in the *map.txt* file produced by *ProcessMap.py*. Then it creates an instance of $PathPlanner$ using the input map. The reason for such a class is so that multiple different planning methods could have been implemented to use the same map and similar functions. The $PRM()$ method is then called with a specified resolution. It creates an instance of a $Graph$. It then generates the specified number of unique random points. These points are also slightly distanced away from obstacles. These points are then added to the graph as milestones which will be explained next.

The $Graph$ class is similar to the usual graph implementations with all the typical methods except vertices or nodes are called milestones here. The $linearNearNeighbours()$ finds fixed-radius (r) near neighbours for each milestone (m) in the input map / grid. It will check if a specified milestone is within range of the current one and if so, check if there are obstacles between them. If there are no obstacles, it will join the two by an edge (represented using adjacency lists). This is how the whole graph is connected. We then perform $dijkstra\_algorithm()$ on the graph which returns a path. Most special cases are handled, for example, when a path is not found or invalid goal coordinates are sent to the program. The path, a list of coordinates, is the output of $FindPath()$.

ROS Turtlebot Execution

The *main.py* script is called initially.

It uses Gazebo $GetModelState$ service to get the current position and angle of the robot. We then get the goal coordinates from standard input. We specify the resolution and radius. We then call $findPath()$. *main.py* handles several different cases where a path is not found. For example, the radius is increased and the algorithm is run a second time. The user can enter another goal after a goal is reached as well.

In *pos.py*, the function $MoveList()$ is called, which takes in path, a list of ordered coordinates, and then applies $MoveTo()$ for each coordinate pair. After all the moves are executed, it will then call the $takePic()$ function, which takes 4 pictures of the surrounding area of the robot. It then calls $detectUtilCart()$ to find the green utility cart in the images. Depending on if the cart is in the images, it will finally publish a 'yes' or 'no' to the *witsdetector* topic. This topic was created by initializing a node an creating a subscriber called *witsdetector*.

$MoveTo()$ gets the current position and angle of the robot. It uses the $euler\_from\_quaternion()$ to get the angle of the robot given its orientation and returns the yaw of the robot. This is used to change the orientation of the robot so that it faces the target location. This function uses *Twist* from *geometry_msgs.msg* to publish the robot's linear and angular velocity to the */mobile_base/commands/velocity* topic.

The $takePhoto$ class subscribes to the */camera/rgb/image_raw* topic. There's a method called $take\_picture()$ which stores the image received from the topic into a file with a specified name. $takePic()$ rotates the robot four times to get a picture from each side of the robot.

Detecting the cart occurs in $detectUtilCart()$. The image is read in using *cv2*. A full scan of the image occurs checking if the RGB values of each pixel matches the RGB of the green cart. If any pixels of this colour exist, then it returns $True$, otherwise it returns $False$ indicating the cart does not exist in the robot's sight.